

LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84

Il 6r

no. 595-600

cop. 2



570.01
Ill 62
no. 596
Cop. 2
Report No. UIUCDCS-R-73-596

Math

A GENERALIZED LEXICAL SCANNER FOR A TRANSLATOR WRITING SYSTEM

by

Albert Cannon Baker, Jr.

October 1973



MAY 9 1974
UNIVERSITY OF ILLINOIS
LIBRARY

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-73-596

A GENERALIZED LEXICAL SCANNER FOR A TRANSLATOR WRITING SYSTEM^{*}

by

Albert Cannon Baker, Jr.

October 1973

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

^{*}This work was supported in part by the National Science Foundation under Grant No. US NSF-GJ-328 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, October 1973.



Digitized by the Internet Archive
in 2013

<http://archive.org/details/generalizedlexic596bake>

A GENERALIZED LEXICAL SCANNER FOR A TRANSLATOR WRITING SYSTEM

Albert Cannon Baker, Jr., M.S.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1973

This is an expository paper that is concerned with a Lexical Scanner for a translator writing system that has been in use at the University of Illinois. Its significant features include a structured, binary-tree symbol table, a parameterized macro expander, and a compile-time flexibility for assigning characters that make up the basic terminal symbols. A comprehensive example of the scanner's operation is also included.

ACKNOWLEDGMENT

I wish to express my deepest thanks to Professor J. R. Phillips for his continuing support as academic advisor and thesis supervisor; to Professor R. S. Northcote, who struck in me the spark of interest in compiler design and defined much of the material for the initial lexical scanner; and to Norma E. Able for her patience through many discussions leading me through the maze of TRANQUIL and the TWS.

Finally, to my wife Carol, thanks for the continued encouragement and motivation, without which the successful completion of these studies would never have been possible.

The Air Force Institute of Technology, Wright Patterson Air Force Base, Ohio, sponsored the author's studies at the University of Illinois.

TABLE OF CONTENTS

	Page
1. INTRODUCTION.	1
1.1 The Translator Writing System.	1
1.2 The Lexical Scanner.	2
2. SCANNER DATA STRUCTURES	5
2.1 General Considerations	5
2.2 BIGTAB - The Symbol Table.	6
2.2.1 Basic Format.	7
2.2.2 Storage for Keywords, Identifiers and String Literals.	8
2.2.3 Storage for Numeric Literals.	9
2.3 The SCAN Descriptor.	12
2.4 MACROTAB - The Macro Text Table.	14
2.4.1 Concept of a Definitional Facility.	14
2.4.2 The TWS Macro Facility.	14
2.4.3 MACROTAB Descriptors.	16
2.5 CHARCLASS - The Character Class Table.	18
3. MAIN SCANNER PROCEDURES	22
3.1 READACARD.	22
3.2 NEXTCHAR	23
3.3 Boolean Procedure TABLESEARCH.	24
3.3.1 Table Lookup.	24
3.3.2 Processing of an Entry Already in BIGTAB.	25
3.3.3 Insertion of a New Entry Into BIGTAB.	26
3.4 Procedures to Build BIGTAB Entries	27
3.5 The Macro Facility	27
3.6 Alpha Procedure SCAN	30
3.7 Integer Procedure SHAKEOUTBIGTAB	31
APPENDIX.	36
A SYNTAX AND SEMANTICS OF SCANNER-DEFINED ITEMS.	36
B SAMPLE PROGRAM	42
LIST OF REFERENCES.	55

LIST OF FIGURES

	Page
1. BIGTAB Basic Format.	7
2. Format of the SCAN Descriptors	13
3. Table Lookup Algorithm	25
4. Examples of Balanced Tree Structures	33
5. Unbalanced Tree Structure.	43
6. Balanced Tree Structure.	44

1. INTRODUCTION

1.1 The Translator Writing System

One of the software efforts that was undertaken for the Illiac IV project at the University of Illinois at Urbana-Champaign was the development of a Translator Writing System which permits the implementation of the various compilers; in particular, the compilers for the Illiac IV problem-oriented languages TRANQUIL [1] and GLYPNIR [2] are the prime examples. The main components of the TWS are:

- a) syntax meta-languages (TWINKLE [7] and TBNF [4]), which are extensions to Backus-Naur Form [8, 9], and in which the syntax of a programming language $\mathcal{L}$ is specified;
- b) a semantics meta-language (Illinois Semantics Language ISL [5]) which is an extension of Burroughs Extended Algol that includes special constructs to manipulate tables, stacks and generate object code; and in which the semantics of a programming language $\mathcal{L}$ is specified;
- c) a basic core for each translator consisting of the lexical scanner, either the skeleton parser for a direct parsing algorithm translated into Algol code or a complete table-driven parser for an interpretive parsing algorithm [4], and miscellaneous auxiliary procedures which are independent of the source and object languages of the translator;
- d) a system consisting of syntax preprocessors to generate either the tables for the interpretive parser or a set of Algol source statements that will parse the language

specified; the ISL translator to translate the ISL extensions into Algol; a program to generate from the outputs of the syntax preprocessor, the ISL translator, and the basic core a complete Algol program which, when compiled by the standard Algol compiler, will be the required translator for the language $\mathcal{L}$ as specified by the syntax and semantics.

1.2 The Lexical Scanner

The function of a lexical scanner in a compiler is to scan characters from a source program, combining one or more of them together to form single terminal symbols when the syntactic recognizer (parser) makes a request for a new symbol. As far as the TWS and the scanner are concerned, the following symbols are deemed to be terminal symbols:

- a) special single characters (\$, : - + =, etc.);
- b) key words in the language, including either reserved identifiers or special identifiers for which a special character (nominally "#") precedes the identifier (BEGIN, #ELSE, etc.);
- c) identifiers (X, CASHIN1STNATIONALBANK, etc.);
- d) string and numeric literals ("THIS IS A STRING", 3.14159, @ 234, etc.).

Internal to the scanner is a powerful parameterized text-type [3] macro expander which has the capability to recognize and store declarations of defined identifiers, and to regurgitate the stored text when the identifier is used subsequently. This facility is transparent to the syntactic recognizer and, except for blockstructure considerations, is transparent to the semantics

as well. Section 2.4 contains a discussion of the data structure of the macros, and Appendix A includes a discussion of the syntax and semantics of the macro generator.

The symbol table used by the scanner is a straightforward binary-tree structure, with disjoint trees for the several terminal symbol classes interleaved within the same table. Binary Coded Decimal (BCD) information is stored in this table packed six characters per 48-bit Burroughs B-5500 word. Section 2.2 contains a complete description of the symbol table.

There is much flexibility built into the scanner to make the resultant compilers both more general and easier to use. Through control card options, the user of the compiler may specify that non-standard symbols be used to define the terminal classes, such as using "8" instead of "@" to mark the exponent part of a numeric literal, or that the key words in the language would be marked by a special symbol, freeing these identifiers for the programmer's use. The inclusion of a macro facility gives the programmer the power to extend the basic language, or to make one language resemble another, or to make his source code appear more readable. For example, if the compiler for an Algol-like language were written using the TWS, one could extend the language at compile time by adding appropriate macro definitions to a program written in it to make it resemble COBOL:

```

DEFINE  ADDING = MEND,
        TO = + MEND,
        COMPUTE = MEND,
        BY = := MEND ;

```

where MEND terminates a macro definition. Then the source language statement:

```

COMPUTE XYZ BY ADDING A TO B;

```

would be compiled as:

$$XYZ := A+B$$

since ADDING and COMPUTE were both defined to be null.

Thus, to reiterate, the main functions of this lexical scanner are to assemble the terminal symbols from the source string, to pass simple representations of those symbols to the syntactic recognizer, to maintain the BCD symbol table, and to perform macro expansion. The data structures behind these functions are the subject of Chapter 2, and a functional description of these functions in terms of the Algol procedures that implement them are the subject of Chapter 3.

2. SCANNER DATA STRUCTURES

2.1 General Considerations

The structure of the internal tables and the algorithms to use them will have a great effect on the speed and efficiency of any program. The lexical scanner is one of the most used procedures in any compiler, and attention must be paid to make it as efficient as possible. In the implementation described here, one of the main considerations is the structure of the language into which the compilers are translated, Burroughs extended Algol for the B-5500. A brief introduction to the B-5500 and its constraints on the Algol language are appropriate here.

The B-5500 [8] is a multiprogramming, multiprocessor computer system. With a limited (32K 48-bit words) main memory, it relies heavily on segmentation of both programs and data to make most effective use of limited memory to service the various programs in the mix. Specifically, programs and arrays are broken down into segments, each no larger than 1024 words. The program segments are stored on the disk. When a program enters the mix to be run, it is assigned a fixed, non-overlayable, contiguous area for a run-time stack and program reference table. This latter contains storage for single variables and descriptors relating to each program and array segment. Then, program and data segments are read off the disk as they are needed, and assigned space in core possibly overlaying previous information from any of the programs in the mix. If the area being overlaid contains only program segments, or array segments that have not been written into, it is simply overwritten; the information is still on the disk. However, an area containing array segments with words that have been changed causes those segments to be written back onto the disk before being overlaid.

The restriction that array segments be no longer than 1024 words is of primary interest here. The segmentation is by array rows - with each row occupying a segment. Thus, no row may be longer than 1024 words. A one-dimensional linear array is one row, so no linear array may be longer than 1024 words. Larger linear tables must be simulated as two-dimensional arrays. For instance, an 8192 word table could be declared with array bounds [0:15, 0:511] so there would be sixteen segments each containing 512 words. When simulating large linear arrays, it is wise to express the range for each subscript to be a power of two in each case in order to be able to access an entry in the table using a single index. In the case above, the column subscript requires exactly four bits, whereas the row subscript requires exactly nine. Thus, given a single 48-bit index I , $I.[35:4]$ would select the proper row, whereas $I.[39:9]$ would select the proper column position within that row (in the partial word notation of Burroughs extended Algol).

2.2 BIGTAB - The Symbol Table

In any compiler, storage for the representations of the terminal symbols must be made. The efficiency of the compiler can be greatly affected by the choice of data structure for the symbol table. The specific functions that must be optimized in the use of the symbol table are, in order of importance:

- a) lookup
- b) insertion
- c) traversing.

Furthermore, separate lists must be maintained for the four classes of multi-character terminal symbols: $\langle *I \rangle$, $\langle *N \rangle$, $\langle *S \rangle$, and $\langle *R \rangle$.

The basic structure BIGTAB was chosen so as to store data as a forest of binary trees, having four trees interleaved within one 8192-word table. The advantages of this approach are:

- a) being naturally linked lists, interleaving the trees in the same table is possible; thus, an identifier entry could be adjacent in the table to a numeric literal; space has to be reserved for only one table;
- b) lookup is fast compared to a linearly linked list;
- c) insertion can be made in the next sequential location in the table with no need to change links already established.

Knuth [6] discusses extensively the characteristics of binary tree structures.

2.2.1 Basic Format

Each tree has a head node in a fixed table location: BIGTAB[1] for identifiers, BIGTAB[2] for numeric literals, BIGTAB[3] for string literals and BIGTAB[15] for key words. This head node is a pointer to the root of its tree. Each BIGTAB entry consists of an entry head plus one to eight data words to store the BCD characters of the text.

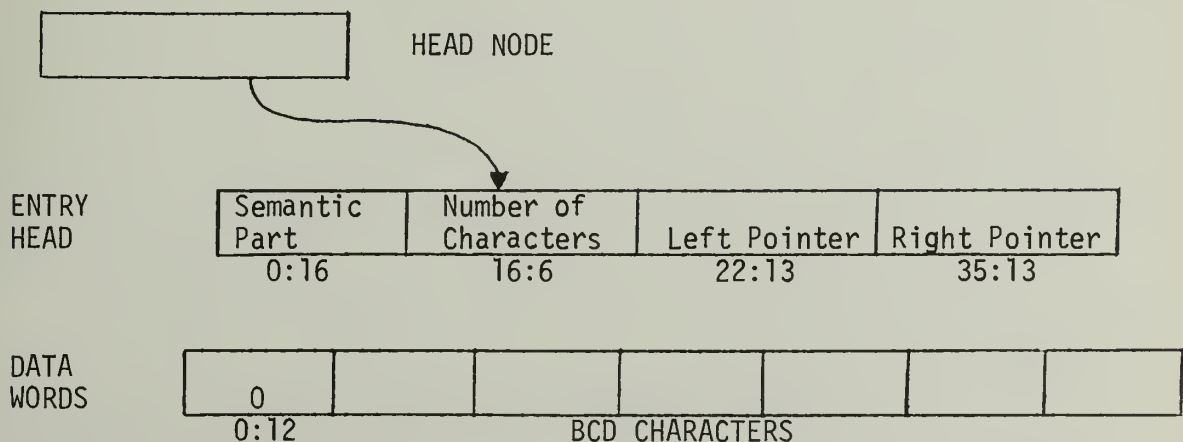


Figure 1. BIGTAB Basic Format

2.2.2 Storage for Keywords, Identifiers and String Literals

The syntax preprocessor will extract from the syntax definition the language specific keywords and non-terminals and place them, linked together in one tree in BIGTAB format, into a disc file called "<LANGNAME>/TABLES", where <LANGNAME> is the name assigned by the compiler designer. At the start of every execution of the TWS-built compiler, this file is read, initializing the run-time BIGTAB. The language non-terminals (such as <BLOCK>, or <ARITHMETIC PRIMARY>) will allow the table-driven parser to provide a trace of the parsing path. The non-terminals are inserted into the table with a leading blank so they will never be recognized as identifiers in the source string. Appendix B lists the TBNF syntax of a simple language DEMALGOL, and gives an example of the initial BIGTAB produced by this syntax.

The scanner assumes as the nominal condition that these keywords will always be preceded by the special symbol "#" (i.e., #BEGIN), and that BIGTAB[15] will point to this initial syntax-preprocessor-built tree. Thus, all occurrences of "#" followed by an identifier will cause reference to this tree. But, by control card option, the nominal condition can be replaced by a reserved word option. In this condition, all occurrences in the source string of all syntax-defined keyword identifiers (i.e., BEGIN) will be reserved to have only the keyword meaning, and BIGTAB[1], the identifier tree, is set to point to the syntax preprocessor-built tree. Thereafter, all identifiers in the source string will be checked against this table, and newly-defined identifiers will be linked into it. The scanner will recognize the presence of a reserved identifier by the fact that the BIGTAB address is within the range of the initial table.

For keywords, identifiers and string literals, the basic format is exactly as specified in section 2.2.1. The only difference among the three

classes is in the use of the semantic part. The BCD characters are stored six per 48-bit word, allowing a maximum of 48 significant characters.

For identifiers, headword bit [1:1] is reserved by the scanner to indicate this identifier is defined as a macro or macro formal parameter. If set, then bits [4:12] point to the address in MACROTAB of the stored text, and bits [2:1] indicate a formal parameter. If the identifier is not defined as a macro, bits [2:13] may be set by the compiler semantic routines as desired. In GLYPNIR [2], as implemented using the TWS, pointers to the semantic IDTAB and the parser MSTACK are inserted in the semantic part.

For keywords, the syntax preprocessor places in the semantic part a unique symbol number for each keyword in the syntax. This allows the parser to consider the keyword as it would a single special symbol.

For string literals, the semantic part is reserved for the semantic routines, typically to point to a literal table.

2.2.3 Storage for Numeric Literals

A numeric literal is a string of characters that carries an inherent semantic value - the specific quantity that this string represents. As this semantic value will be variable and machine dependent, the TWS will not convert these literals to an internal machine representation, but rather transform the literal string to a normalized, consistent BIGTAB entry, with enough analysis performed on the source string to make the semantic conversion of the numeric literal to internal representation relatively straightforward for the semantic part of the compiler.

In BIGTAB, the same basic header word and data word structure applies here as in the identifier, keyword and string literal tables. The semantic part of the header word can be used, as in the TRANQUIL compiler implemented

using the TWS, to store a literal type, and a pointer to a semantic table. But, in the data words, quite a different structure is used. Instead of the BCD characters packed six per 48-bit word, the full eight character capacity of each word is used, with the first two characters in the first data word being used to describe certain semantic attributes of the numeric literal:

ENTRY
HEAD:

Semantic Part	Number of Characters	Left Tree Pointer	Right Tree Pointer
0:16	16:6	22:13	35:13

DATA
WORDS:
(1-8)

Char 0	Char 1	Char 2	Char 3	Char 4	Char 5	Char 6	Char 7
0:6	6:6	12:6	18:6	24:6	30:6	36:6	42:6

The first two characters (12 bits) of the first data word have the following values:

- 0:1 - Unused, always zero
- 1:1 - Base indicator
 - =0 Decimal numeral, base 10
 - =1 Nondecimal numeral, base 2 to 36
- 2:1 - Numeric type
 - =0 Integer
 - =1 Real
- 3:1 - Sign of exponent
 - =0 Positive
 - =1 Negative
- 4:2 - Number of exponent digits (I)
 - 0-3 (i.e., exponent $0-999_{10}$)
- 6:6 - Number of mantissa digits (N).
 - $1-62_{10}$

There follow "N" characters of the mantissa; followed by "I" by characters of the exponent (for real type numeric literals only); followed by one character containing the base (for nondecimal base numeric literals only), range $2-36_{10}$. The last data word is zero filled.

For numeric literals containing a radix point, the mantissa is normalized, that is the exponent is recomputed as though the radix point is to the right of the rightmost mantissa digit.

For nondecimal bases, there must be provision for up to 36 different digits. The scanner considers 0-9 and A-Z as the 36 digits. The internal character code for the decimal digits 0-9 exactly correspond to the "digit value" 0-9. But this is not true for the alphabetic letters. To correct for this, the input alphabetic character will be converted to a true digit value in the range 0-35 for storage in the data words. This is accomplished by subtracting a bias from the character code, depending on the letter:

A-I subtract 7
 J-R subtract 14
 S-Z subtract 22.

Consider as an example, the hexadecimal numeric literal 3A42E.569@-354(16). The semantic descriptor would be composed as follows:

0:1 =0
 1:1 =1, nondecimal base
 2:1 =1, real
 3:1 =1, negative exponent
 4:2 =3, 3 digits of exponent
 6:6 =8, 8 digits of mantissa

This produces for the first 12 bits 0111 111 001 000, or as six bit characters,

"←8". This will produce the following data words:

Data word 1: ← 8 3 # 4 2 > 5

Data word 2: 6 9 3 5 7 + 0 0

Note that the exponent has been changed from 354 to 357, representing the normalization; that the base is represented by "+", or 16_{10} ; that the second word is padded with zeros; that hex A (character code 17) is converted into "#" (character code 10); and that hex E (character code 21) is converted into ">" (character code 14).

The simple decimal integer 1 would be converted for storage to:

0:1 =0

1:1 =0, decimal base

2:1 =0, integer

3:1 =0, positive exponent

4:2 =0, no exponent part

6:6 =1, 1 mantissa digit.

This produces the six bit characters "01", and the following data word:

Data word 1: 0 1 1 0 0 0 0 0

2.3 The SCAN Descriptor

The ALPHA procedure SCAN is called by the parser (and recursively from within the scanner itself) when a new terminal symbol is required. The 48-bit value assigned to SCAN as a function is referred to as the SCAN descriptor, and has the format as shown in Figure 2.

For keywords <*R>, the symbol number is assigned by the syntax pre-processor, starting with 66_{10} . Special single characters have a symbol number equal to their internal 6-bit character code, thus varying from 0_{10} (numeral

zero) to 63_{10} ("). This allows the keywords and the special single characters to be considered the same in the parsing routines.

<*I>, Identifiers:

Unused	Class =1	BIGTAB Semantic Part	Pointer to BIGTAB	Class =1	Pointer to BIGTAB
0:2	2:4	6:12	18:13	31:4	35:13

<*N>, Numeric Literals:

Unused	Class =2	BIGTAB Semantic Part	Pointer to BIGTAB	Class =2	Pointer to BIGTAB
0:2	2:4	6:12	18:13	31:4	35:13

<*S>, String Literals:

Unused	Class =3	BIGTAB Semantic Part	Pointer to BIGTAB	Class =3	Pointer to BIGTAB
0:2	2:4	6:12	18:13	31:4	35:13

<*R>, Keywords:

Unused	Class =15	Symbol Number	Pointer to BIGTAB	Class =15	Symbol Number
0:2	2:4	6:12	18:13	31:4	35:13

Special Single Characters:

Unused	Class =15	Symbol Number	Symbol Number	Class =15	Symbol Number
0:2	2:4	6:12	18:13	31:14	35:15

Figure 2. Format of the SCAN Descriptors

For keywords <*R>, the symbol number is assigned by the syntax pre-processor, starting with 66_{10} . Special single characters have a symbol number equal to their internal 6-bit character code, thus varying from 0_{10} (numeral

zero) to 63_{10} ("). This allows the keywords and the special single characters to be considered the same in the parsing routines.

2.4 MACROTAB - The Macro Text Table

2.4.1 Concept of a Definitional Facility

Many compilers in current use (B-5500 ALGOL, IBM PL/I, JOVIAL, etc.) have a definition facility--that is capability to define compile-time procedure-like constructs. One can compare a text-type definition or macro facility with a run-time procedure construct as follows:

A procedure

- a) is considered syntactically as a complete <statement> or <primary>;
- b) produces one set of machine code that may be executed by jumps and parameter linkages from different parts of the main program.

A macro

- a) may be an incomplete syntactic fragment composed of a sequence of terminal symbols;
- b) produces a separate set of machine code for each invocation;
- c) is strictly a compile-time device that is transparent to the parsing and semantic portions of the compiler.

2.4.2 The TWS Macro Facility

As the TWS was developed using Burroughs B-5500 ALGOL, it became apparent that the definition facility implemented on this compiler made the compiler easier to use, and actually allowed local extensions to be implemented in a rather straightforward manner. Therefore, as a practical matter a similar

parameterized macro expander was included as a part of the core compiler for all TWS-written compilers.

The storage scheme selected was to store the macro text as the entire SCAN descriptor, with two header words for each definition. Some elements of the storage scheme are:

- a) one 48-bit word per terminal symbol;
- b) the scope of identifiers (a semantic concept) used in the macro text will be defined at the point in the program where the macro is declared, since the SCAN descriptor includes the BIGTAB semantic part at the time the macro was declared.
- c) accessing the pre-stored macro text by the scanner may be faster than scanning the text from the source string--as the time-consuming assembling of the characters into the numeric strings and identifier strings, and table lookup in BIGTAB is performed only once, no matter how many times the macro is invoked;
- d) block structure considerations are made to allow an identifier to be defined, for example, as a label in one block and redefined as a macro in an inner block with the old semantic definition being restored upon block exit;
- e) a defined identifier may be redefined within the block in which it was declared, in which case the new text will replace the old text for subsequent invocation (this implies that the macro declaration does not necessarily have to be placed in the block head for a block-structured language);
- f) no parsing or syntax checking of the text is made until the defined identifier is invoked;

- g) defined identifiers (i.e., calls on other macros) may occur anywhere within the macro text, but the value is defined at the point of the macro declaration;
- h) defined identifiers may occur anywhere within the actual parameter part of any macro call;
- i) no recursion--i.e., one macro directly or indirectly calling itself--is permitted.

Appendix A describes the detail of the syntax and semantics of the elements of the macro facility.

2.4.3 MACROTAB Descriptors

As the macro text is processed by the scanner from the DEFINE declaration, two header words are set up in MACROTAB:

WORD1:	Unused 0:6	Address of Return Descriptor 6:12	Address of Actual Parameter Table 18:12	Number of Parameters 30:12	Unused 42:6
WORD2:	BIGTAB Semantic Part of Defined Identifier 0:16	Pointer to BIGTAB Address of Defined Identifier 16:13	Link to Previous MACROTAB entry 29:12	Block Nesting Level 41:7	

The text is then scanned (by SCAN) into the table, one word per terminal symbol. If a defined identifier is encountered in the source string, a special macro call descriptor is inserted into the table. If a formal parameter is encountered, a special formal parameter descriptor is inserted. Finally, at the end of the text, a return descriptor is inserted:

SCAN DESCRIPTOR:	Unused	Class	Symbol #	BIGTAB Pointer for <*I> <*N> <*S>	Class	BIGTAB Pointer
	0:2	2:4	6:12	18:13	31:4	35:13

MACRO CALL DESCRIPTOR:	Unused	Class =8	Pointer to Called Macro	Address of Where to Continue after the Call	Contents of Called Macro's Return Word
	0:2	2:4	6:12	18:12	30:12

RETURN DESCRIPTOR:	Unused	Class =9	Where to Continue Processing upon Return; =0 Means Outermost Macro	Address of Macro Call Descriptor
	0:2	2:4	6:12	18:12

FORMAL PARAMETER DESCRIPTOR:	Unused	Class =10	Address of Macro Header Word	Parameter Number
	0:2	2:4	6:12	18:12

When the macro is invoked, the actual parameters must be stored, in a manner similar to the macro itself, as scan descriptors. In addition to the stored text, for each actual parameter, there will be one return descriptor as described above plus one parameter address and length word for each two actual parameters.

ADDRESSES AND LENGTH DESCRIPTOR:	First Length	First Address	Second Length	Second Address
	0:12	12:12	24:12	36:12

As the formal parameters used within the macro definitions are strictly local, provision has been made to use the high-order end of the macro table (from location 4095 down) as temporary storage for the semantic part of the parameter identifiers during scanning of the macro text. This semantic part is then restored when the mend terminating the definition is scanned:

FORMAL PARAMETER SAVE WORDS:	BIGTAB Semantic Part	BIGTAB Pointer
	0:16	16:13

2.5 CHARCLASS - The Character Class Table

The term "terminal head symbol" refers to the first character of a terminal symbol. The scanner needed a way to determine from the terminal head symbol what class of terminal symbol was to follow. For example, a decimal digit, radix point or exponent sign will indicate a numeric literal must be formed from the following characters. Similarly, a string quote indicates a string literal follows. For this and other decision points in the scanner, a table of character classes has been established, assigning to each six-bit BCD character a bit string:

Character Class	Class Value	CHARCLASS Bit Positions							
		41	42	43	44	45	46	47	
Digits 0-9	58	0	1	1	1	0	1	0	
Special Keyword Delimiter (<u>#</u>)	4	0	0	0	0	1	0	0	
Numeric Literal Exponent Delimiter (<u>@</u>)	18	0	0	1	0	0	1	0	
Radix Point (<u>.</u>)	34	0	1	0	0	0	1	0	
Numeric Literal Base Delimiter (<u>_</u>)	64	1	0	0	0	0	0	0	
String Quote (<u>"</u>)	3	0	0	0	0	0	1	1	
All Other Special Symbols	0	0	0	0	0	0	0	0	
Letters A-I	89	1	0	1	1	0	0	1	
Letters J-R	105	1	1	0	1	0	0	1	
Letters S-Z	121	1	1	1	1	0	0	1	

Table 1. CHARCLASS Table

When certain lexical decisions must be made about a character in the source string, a branch is made, indexed by some subset of the bits in the CHARCLASS entry.

When the scanner is ready to search for a new terminal symbol, a branch is made on bits [45:3] of the CHARCLASS of the terminal head symbol, with the following results:

<u>[45:3] Value</u>	<u>Characters in This Class</u>
1	Letters A-Z. Identifier follows.
2	@, ., Digit 0-9. Numeric literal follows.
3	". String literal follows.
4	#. If the special word option for keywords was chosen, a language keyword follows.
0	All other special characters. Process as a special single character.

During the assembly of a numeric literal, when it is known the terminal head symbol has bits [45:3]=2, a further branch is made on bits [42:2], with the following results:

<u>[42:2] Value</u>	<u>Characters in This Class</u>
1	@. Exponent delimiter.
2	.. Radix point.
3	Digits 0-9.

These correspond to special processing depending on the numeric type.

Later, during assembly of the interior symbols of the numeric literal, a branch is made on bits [41:3] of the incoming source symbol, with the following results:

<u>[41:3] Value</u>	<u>Characters in This Class</u>
1	@. Branch to exponent logic.
2	.. Branch to fractional digit logic.
3	Digits 0-9. Assemble in normal manner.
4	(. Branch to base logic.
5	A-I. Branch to logic to convert character codes 17-25 to true digit values 10-18.
6	J-R. Branch to logic to convert character codes 33-41 to true digit values 19-27.
7	S-Z. Branch to logic to convert character codes 50-57 to true digit values 28-35.
0	All other special symbols. Terminate numeric Literal processing.

During assembly of the alphanumeric internal characters of an identifier, CHARCLASS bit [44:1] is used to indicate an alphanumeric character.

<u>[44:1] Value</u>	<u>Characters in This Class</u>
1	Digits 0-9, letters A-Z. Assemble into the identifier.
0	All other characters. Terminate identifier processing.

This procedure of using the internal character code of the source characters to index a table of character classes allows the TWS-supplied control card options below to be easily implemented:

ALPHABETIC <u>a</u>	Let <u>a</u> have CHARCLASS.[44:4]=9. Thus <u>a</u> can be an identifier terminal head symbol ([45:3]=1) or an internal identifier symbol ([44:1]=1).
ALPHANUMERIC <u>b</u>	Let <u>b</u> have CHARCLASS.[44:1]=1. Thus <u>b</u> can be an internal identifier symbol.
RSWD	Choose the reserved word option for keywords. Let <u>#</u> have CHARCLASS = zero.
SPWD <u>c</u>	Retain the nominal special word option for keywords, but designate <u>c</u> to be the delimiter by setting CHARCLASS <u>c</u> to 4, and reset CHARCLASS <u>#</u> to zero.
EXPONENT <u>d</u>	Change CHARCLASS <u>d</u> to 18; reset CHARCLASS <u>@</u> to zero.

Specifying "ALPHANUMERIC - " on a control card would allow a COBOL-like identifier CASH-IN-FIRST-NATIONAL-BANK.

3. MAIN SCANNER PROCEDURES

This chapter will give a functional description of major parts of the lexical scanner in terms of the ALGOL procedures that perform the various functions.

3.1 READACARD

This procedure defines the basic format of the source program cards accepted by the TWS scanner.

- a) The source program card images are read from the disc file "SOURCE" as 80-character records, into an array CARDBUF[0] to CARDBUF[9], appropriately recognizing the end-of-file condition.
- b) The text in card image columns 1-72 is then transferred into another array CHARBUF[1] to CHARBUF[72], one BCD character per word.
- c) The card image is analyzed to identify leading and trailing blanks, setting items "FCR" as the column with the first non-blank character, "LCR" as the last non-blank character, and "NCR" as the moving character pointer initially set equal to "FCR".
- d) The card image counter "CARDCOUNT" is incremented by one, with the current value placed in CARDBUF[10] for printing as columns 81-89 of the card image.
- e) The number in columns 72-80 of the card image is translated to internal form, and made available to the rest of the compiler procedures as "CARDNUM". If this field is blank on the first card image in the source stream, "CARDNUM" will be

- subsequently set equal to "CARDCOUNT" for each card image.
- f) If column one is "\$", the card image is printed on printer backup disc file "LINE", and control is passed to the procedure "CONTROLCARD" for analysis of the control card information.
 - g) If the card image was not a control card, and if the "PRINT" or "LIST" control card options had been chosen previously, the contents of array CARDBUF, including the inserted card counter, is printed on file "LINE".

Thus, externally, the following are the user-oriented TWS features implemented through READACARD:

- a) control cards start with a "\$" in column one;
- b) source text occupies columns 1-72;
- c) columns 72-80 may contain a card count, that will be made available, if the semantic routines store it, for program traces.

3.2 NEXTCHAR

This procedure provides other scanner routines with text one character at a time. Its output is a variable NXTCHR containing the six-bit BCD code of the character scanned in the source string. In addition, the following functions are performed:

- a) If a "%" is detected anywhere on the card image, the rest of the card image is ignored. This is the basic COMMENT capability provided by the TWS. An ALGOL-like facility may be implemented by the semantics, with the caveat that all the text in the comment would be scanned, with all words and

numbers placed into BIGTAB.

- b) When not internal to a string literal, multiple blanks in the source stream are reduced to a single blank for parsing purposes.

3.3 Boolean Procedure TABLESEARCH

Prior to the execution of TABLESEARCH, other procedures will have been run to assemble from the source string one of the classes <*I>, <*R>, <*S>, or <*N> of multi-character terminal symbols into SYMBUF[0] to SYMBUF[7], in the format of the BIGTAB data words described in section 2.2. The value of the procedure will be TRUE if a successful BIGTAB table lookup has been made, meanwhile setting the global variable NEXTSYM to be the SCAN descriptor (see section 3.2). The value is set to FALSE if a macro definition or call is encountered, indicating the main SCAN procedure must then either assemble further text from the source string, or obtain SCAN descriptors from the macro table.

The TABLESEARCH procedure has three main portions that will be described in detail: table lookup in BIGTAB, processing of an entry already in BIGTAB, and insertion of a new entry into BIGTAB.

3.3.1 Table Lookup

BIGTAB is a straightforward binary tree structure. The basic algorithm below does not reflect the complication in the scanner that is required by the fact that entries to be compared may be of different lengths, extending over one to eight words.

Given SYMBUF[0] to SYMBUF[7] containing an entry in BIGTAB format;
 TYP, the terminal symbol class (1, 2, 3, 15) of the entry;
 NWDCH, the length of the SYMBUF entry;
 LEFTPOINTER and RIGHTPOINTER referring to pointers in the
 BIGTAB entry head

```

L1(Find root)           : Set ENTRYPTR ← BIGTAB[TYP]
L2(Test if link null)   : Is ENTRYPTR = 0? Yes, go to EXIT1.
L3(Compare SYMBUF       : If SYMBUF = BIGTAB entry, and length of SYMBUF =
  with BIGTAB)           length of BIGTAB entry, then go to EXIT2.
L4(SYMBUF ≠ BIGTAB)     : If SYMBUF < BIGTAB entry,
                        set ENTRYPTR ← BIGTAB[ENTRYPTR].LEFTPTR,
                        set K ← 1, go to L2.
L5(SYMBUF > BIGTAB)     : Set ENTRYPTR ← BIGTAB[ENTRYPTR].RIGHTPOINTER,
                        set K ← 2, go to L2.
EXIT1                   : Entry not in BIGTAB. See section 3.3.3.
EXIT2                   : Entry already in BIGTAB, ENTRYPTR is location
                        of head node. See section 3.3.2.
  
```

Figure 3. Table Lookup Algorithm

3.3.2 Processing of an Entry Already in BIGTAB

If the symbol scanned in the source string is already in BIGTAB, three cases must be distinguished:

- a) The keyword DEFINE is encountered. A macro definition follows. Process the text into MACROTAB format (see section 2.4). Exit TABLESEARCH, indicating an unsuccessful table lookup.

- b) A defined identifier indicating a macro call is encountered. Process the actual parameters of the call (see section 2.4) if any, change the scan mode to indicate subsequent scan descriptors are to come from the macro table. Exit TABLESEARCH, indicating an unsuccessful table lookup.
- c) A normal entry is encountered. Build the scan descriptor NEXTSYM according to the symbol class. Exit TABLESEARCH, indicating successful table lookup.

3.3.3 Insertion of a New Entry Into BIGTAB

When the symbol scanned in the source string is not found in the BIGTAB table lookup, it must be then inserted into the symbol table:

- a) The head word is created, inserting only the number of characters. The semantic routines will set the semantic part, and both right and left tree links will be empty when the entry is created.
- b) The head word and the data word(s) are inserted into the table in the next available sequential location. A check is made to insure that the complete entry will fit into one array row, as to split elements of one symbol across array rows would cause undue overhead due to array row segmentation in the B-5500.
- c) It was found that the entry was not in BIGTAB when either the right or the left of the entry head in location ENTRYPOINTER was null. If in L4, K was set to one, then the left pointer of the entry head at location ENTRYPOINTER must be set to point to this new entry address. Otherwise, in L5, K was set to two so the right pointer must be set to

point to the new entry.

- d) Build the scan descriptor NEXTSYM according to the symbol class. Exit TABLESEARCH, indicating a successful table lookup.

3.4 Procedures to Build BIGTAB Entries

There exist three major procedures, NUMERICLIT, STRINGET, and ALPHAGET to assemble into SYMBUF 0 to SYMBUF 7 the numeric literal, string literal and identifier data types. They are functionally described by their outputs in section 2.2.

3.5 The Macro Facility

Section 2.4 describing the data storage for the macro text gives an adequate functional description of the procedures PROCESSMACRODECLARATION, MACROINVOCATION, and PROCESSMACROACTUALPARAMETERPART. This section will discuss the procedure GETDESCRIPTORFROMMACROTAB, to illustrate how it "executes" the descriptors placed in the macro table by the other procedures.

The major concept in designing this descriptor-based macro system that would allow nearly arbitrary text in the parameters of the macro invocation, and that would further allow arbitrary (except recursive) macro invocations either within the text or within the actual parameter was the concept that the descriptors could be considered as "instructions" directing the flow of data from the table, that would be "interpreted" by the Alpha procedure GETDESCRIPTORFROMMACROTAB. If a formal parameter or call on another macro is detected, during scanning of the text in the declaration, a special "jump instruction" is placed in the sequential macro table to direct the flow of data. At the end of the macro text itself, and at the end of an actual parameter, a "return" word is inserted - to direct the flow back to the point where it was interrupted.

The procedure is called from the SCAN procedure when a new symbol is needed in SCANMODE 5. The global variable NEXTMACRO contains the macro table address of the next sequential entry to be chosen. The macro table entry at this address is examined and "executed", depending on the class field, bits [2:4] of the entry:

<u>Class Value</u>	<u>Action</u>
1, 2, 3, 15	<u>Normal SCAN descriptor.</u> Set GETDESCRIPTORFROMMACROTAB $\leftarrow$ MACROTAB [NEXTMACRO]. Increment NEXTMACRO by one. Exit procedure.
8	<u>Macro invocation descriptor.</u> 1) Set up called macro's return descriptor. Return location is either NEXTMACRO + 1 or the address following the actual parameter table. This location has been inserted in bits [18:12] of the macro invocation descriptor by the procedure PROCESSMACRODECLARATION. 2) Set in the <u>called</u> macro's entry head word one the address of the actual parameter table. If parameters are present, their location will be NEXTMACRO + 1. 3) Set NEXTMACRO to the first word of the called macro's text, located immediately following the second header word. 4) Branch to code to examine a new macro table entry.
9	<u>Return descriptor.</u> Either a complete macro call or an actual parameter has been "executed". Consider the following two cases: 1) Return address is zero. This is true only for an outermost macro call. Set GETDESCRIPTORFROMMACROTAB $\leftarrow$ 0,

Class ValueAction

set SCANMODE $\leftarrow$ 0. Set PTMACROTAB (the pointer to the next available location for insertion of next text) to the value it had at the time the defined identifier was scanned in the source string. This will "erase" the actual parameters stored for this call. Exit from the procedure.

2) Return address is not zero. Set NEXTMACRO $\leftarrow$ return address, branch to code to examine a new macro table entry.

10

Formal parameter.

Extract from descriptor bits [6:12] the address of the macro head word one. From bits [18:12] of the head word, extract the location of the actual parameter table as set when the actual parameters were scanned. From bits [18:12] of the formal parameter descriptor, extract the parameter number. Determine from the addresses and lengths descriptor in the actual parameter table the location of the specific actual parameter needed, as well as the address of its return descriptor. Set the actual parameter return address to NEXTMACRO + 1. Set NEXTMACRO $\leftarrow$ actual parameter address. Branch to code to examine a new macro table entry.

Note that none of the "special" descriptors cause output from the procedure, but just a redirecting of the flow, followed by "execution" of the descriptor in the new location. Note also that this procedure will work on either empty macros or empty parameters. In both cases, the stored text will be simply a return descriptor.

3.6 Alpha Procedure SCAN

SCAN is the procedure that controls the actions of all the other procedures mentioned above. It is the prime interface with the parsing routines, and is called when a new terminal symbol is required. The procedures to build the macro table are declared in the SCAN procedure head, and thus call SCAN recursively to obtain the descriptors to store in the macro table.

The value of SCAN as a function is normally the scan descriptor as discussed at length in section 2.4. In SCANMODE 4, its value will be the contents of SYMBUF[0].

There are several modes of operation of SCAN, depending on the way the source characters are to be assembled. Setting of the global variable SCANMODE prior to call of SCAN will cause one of the following actions to be taken:

<u>SCANMODE</u>	<u>Action Taken by the Scanner</u>
0	Normal operational mode. Ignore all embedded blanks outside of string literals. Return normal SCAN descriptors.
1	As in SCANMODE 0, but reduce adjacent embedded blanks to one blank and report as a single special symbol.
2	Scan the text between FCR and LCR. Return a descriptor on each character in the source string as a single special character SCAN descriptor, but ignore blanks.
3	As SCANMODE 2, but reduce adjacent blanks to one and report.
4	As SCANMODE 0, but return contents of SYMBUF[0] - i.e., the first BCD characters of the terminal symbol - as the SCAN descriptor. Do not look up or enter the symbol in BIGTAB.
5	Fetch SCAN descriptor from the macro table.

When in SCANMODE 0, 1 or 4, a branch is made on CHARCLASS[45:3] of the terminal head symbol to define whether an identifier, a numeric literal, a string literal, a special word keyword or a single special symbol follows. Based on the specific branch made, the terminal symbol is assembled in the proper format into the array SYMBUF. In SCANMODE 0 or 1, a BIGTAB table lookup is performed, obtaining the BIGTAB semantic part and address of the symbol. With this information, the SCAN descriptor is assembled.

3.7 Integer Procedure SHAKEOUTBIGTAB

It was noticed when working with the initial BIGTAB produced by the 1969 version of TRANQUIL that there was quite a large imbalance in the tree structure. Specifically, the initial BIGTAB contained 198 entries consisting of 109 keywords and 89 language terminals. A reflection on the properties of binary trees shows that in the worst case, all nodes could be strung out", requiring 198 levels, and in the best case, eight levels ($\lceil \log_2 198 \rceil$). The importance in the number of levels is in the speed of lookup--the more levels to the tree, the more comparisons that must be made to find an entry in the tree.

On the 198-entry tree actually produced by the syntax preprocessor, the level number of the nodes varied from one (for the head of the tree) to eighteen. The average level of all 198 nodes was nine. For comparison, a fully balanced binary tree with eight levels could contain 255 nodes, will have a maximum level of eight, and an average level of 7.03.

An algorithm was developed by this author that will balance the tree structure of any input BIGTAB-type tree, modifying the left and right tree pointers, but leaving all nodes in the same locations as previously. The maximum level of the balanced tree will be $\lceil \log_2(N-1) \rceil$, where N is the total number

of nodes in the tree to be balanced.

Before the algorithm is discussed in detail, some observations can be made about the structure of the balanced tree. Given the nodes in lexical order in the sequential table TAB[1] to TAB[N], the tree developed by this algorithm will have all odd TAB entries, i.e., TAB[1], TAB[3], TAB[5], etc., as terminal nodes-with both left and right tree pointers null. Conversely, all even TAB entries, i.e., TAB[2], TAB[4], etc., will have at least one non-null link. Furthermore, as the tree is "grown", the left sub-tree of any node will always be complete. If the tree is not full, it will be the right sub-trees that will be partially empty. Figure 4 illustrates these points.

The essence of the algorithm is to order the nodes into a linear list TAB, and then to visit each node on each level sequentially, from left to right, computing and setting the new BIGTAB tree links as each node is visited. To control sequencing of the algorithm, a queue is constructed, being initialized both front and rear with the new head node. When the right and left tree links are computed for a node, the TAB address of these sub-nodes are inserted into the rear of the queue. This results in the visit of all nodes in a certain level before progressing to the next lower level.

Use of this procedure has been made a control card option. If "BALANCE" appears on a control card, the initial BIGTAB is balanced. Thereafter, the procedure may be called from procedure TABLESEARCH, whenever a BIGTAB array row fills up.

On a series of benchmark tests using a 1943-card-image input deck, using the 1969 TRANQUIL BIGTAB, the balancing added about 2.8 seconds to the two minute total scan time. But use of the balanced BIGTAB was able to increase throughput of the scanner between six and seventeen percent over that using the unbalanced BIGTAB.

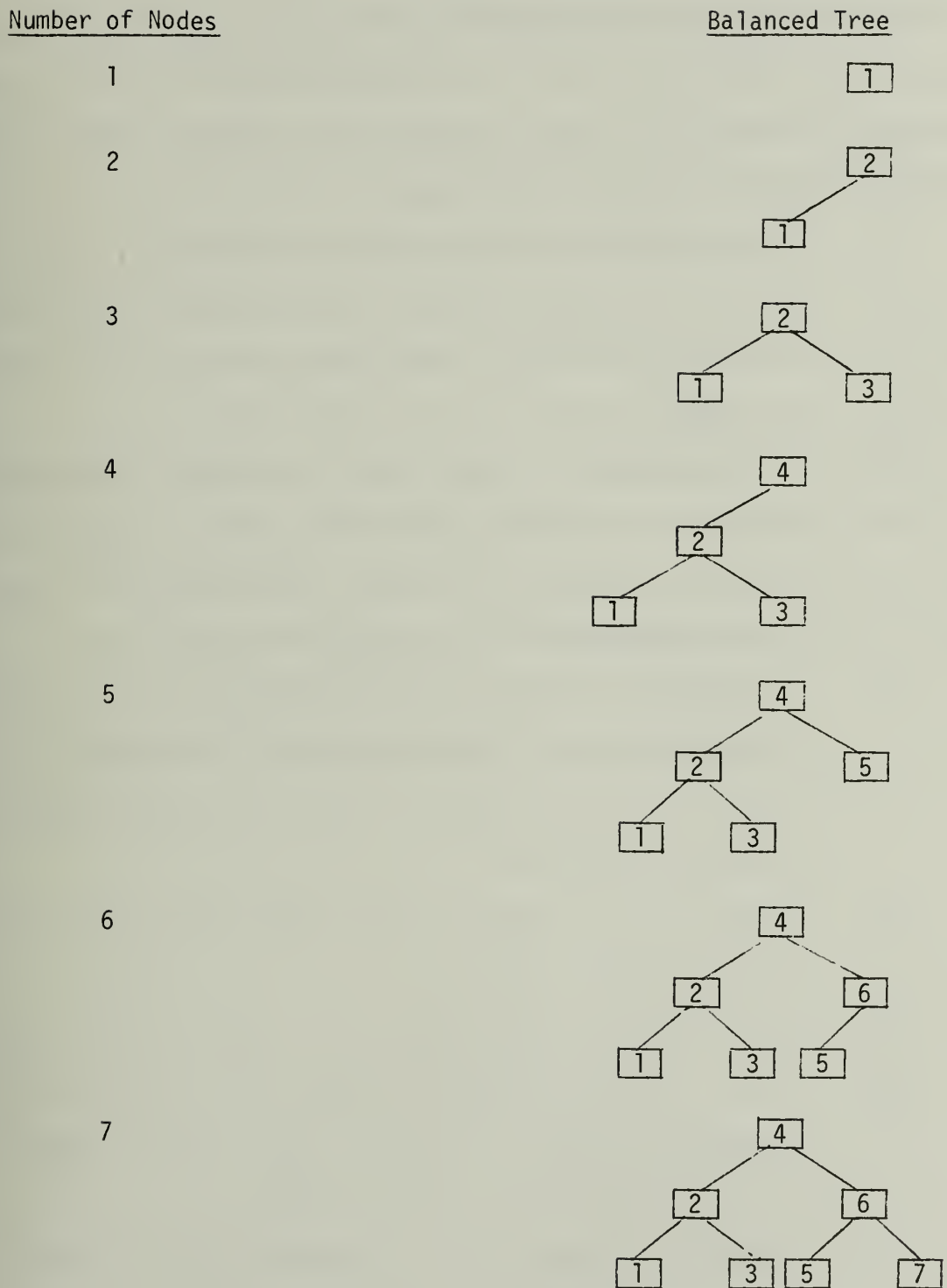


Figure 4. Examples of Balanced Tree Structures

ALGORITHM B - Balance the Tree Structure

Let BIGTABWORD be considered a pointer with bits [22:13] as a left pointer field, and bits [35:13] as a left pointer field. Let each TAB entry have three fields: bits [15:10] as the queue link, bits [25:10] as the delta field, and bits [35:13] as the BIGTAB address.

- B1 Traverse the tree in postorder (see Algorithm T, Knuth [6], page 317), using an auxiliary stack, placing the BIGTAB addresses of the nodes visited into a sequential array TAB[1] to TAB[N].

- B2 Find the size of the smallest fully balanced tree that has less than or equal to N nodes. Let I be this number, with a value 1, 3, 7, 15, 31, $2^m - 1 (m \geq 1)$. The exponent m is the maximum number of levels in the balanced tree.

- B3 Set SHAKEOUTBIGTAB to be the BIGTAB address of the root of the balanced tree. This will be found in TAB[$\lceil I/2 \rceil$], for example, if I=15, TAB[8] will contain the BIGTAB address of the root of the tree.

- B4 Set $F \leftarrow R \leftarrow P \leftarrow \lceil I/2 \rceil$, the root of the tree. Set the delta field of TAB[P] $\leftarrow P/2$.

- B5 Is $F=0$? (output queue exhausted?). Yes, go to B13 (exit).

- B6 Set $P \leftarrow F$ (front of queue). Compute right and left tree pointers for node P. Set DELTA $\leftarrow$ delta field of TAB[P]. Set Q $\leftarrow$ BIGTAB address stored in TAB[P].

- B7 If DELTA = 0, then node P is a terminal node with both pointers zero. Set BIGTABWORD to zero. Go to B12.

- B8 DELTA $\neq 0$. Compute left tree pointer. Set LINK $\leftarrow P - \text{DELTA}$. Set left part of BIGTABWORD to the BIGTAB address field of

TAB[LINK]. Set the delta field of TAB[LINK] to DELTA/2 (as it is on the next lower level), and insert LINK into the rear of the output queue.

- B9 Compute the right tree pointer. Set $LINK \leftarrow P + DELTA$. Since the right sub-tree may be incomplete, recompute DELTA. If $LINK > N$, then set $DELTA \leftarrow DELTA/2$, go to B9.
- B10 If $DELTA = 0$, then the right sub-tree is empty. Set the right part of BIGTABWORD to zero. Since there is no right sub-tree, nothing needs to be put into the output queue. Go to B12.
- B11 $DELTA \neq 0$. Set right part of BIGTABWORD to the BIGTAB address field of TAB[LINK]. Set DELTA/2 into the delta field of TAB[LINK]. Insert LINK into the rear of the output queue.
- B12 Set $BIGTAB[Q].[22:26] \leftarrow BIGTABWORD$. Set F to the next node from the front of the queue. Go to B5.
- B13 Exit procedure. Entire tree is balanced.

APPENDIX A

SYNTAX AND SEMANTICS OF SCANNER-DEFINED ITEMS

<*I>, the Identifier MetaclassTBNF SYNTAX

<LETTER>	::= A B C D ... Z
<DECIMAL DIGIT>	::= 0 1 2 3 4 5 6 7 8 9
<ALPHABETIC CHARACTER>	::= <LETTER> <ALPHABETIC CONTROL CARD OPTION>
<ALPHANUMERIC CHARACTER>	::= <ALPHABETIC CHARACTER> <DECIMAL DIGIT> <ALPHANUMERIC CONTROL CARD OPTION>
<*I>	::= <ALPHABETIC CHARACTER> <*I> <ALPHANUMERIC CHARACTER>

SEMANTICS

All identifiers are limited to 48 characters in length. Identifiers may not extend over card-image boundaries. The standard syntax of <ALPHABETIC CHARACTER> or <ALPHANUMERIC CHARACTER> may be augmented by control card option (see section 2.5).

<*R>, the Keyword MetaclassTBNF SYNTAX

<*R>	::= <*I> "#" <*I>
------	-------------------

SEMANTICS

If the "RSWD" control card option is selected, a simple identifier may be used as a keyword - and cannot be re-declared by the compiler-user. The nominal state is the "SPWD" or special word option, where a "#" must precede the syntax-defined keyword identifier for it to have keyword meaning. The

programmer may then choose freely the identifiers he uses with no fear of encountering reserved identifiers he did not know about.

<*N>, the Numeric Literal Metaclass

TBNF SYNTAX

<DECIMAL INTEGER>	::= <u>list</u> <DECIMAL DIGIT>
<DIGIT>	::= <LETTER> <DECIMAL DIGIT>
<INTEGER>	::= <DECIMAL DIGIT><DIGIT>*
<RADIX POINT>	::= "."
<FRACTIONAL PART>	::= <RADIX POINT><DIGIT>*
<EXPONENT DELIMITER>	::= "@"
<EXPONENT PART>	::= <EXPONENT DELIMITER>[+ -]? <DECIMAL INTEGER>
<BASE LEFT DELIMITER>	::= "("
<BASE RIGHT DELIMITER>	::= ")"
<BASE PART>	::= <BASE LEFT DELIMITER><DECIMAL INTEGER> <BASE RIGHT DELIMITER>
<REAL>	::= [<INTEGER> <FIXED POINT>]? <EXPONENT PART> FIXED POINT>
<FIXED POINT>	::= <INTEGER><RADIX POINT><DIGIT>* <RADIX POINT> <u>list</u> <DIGIT>
<*N>	::= [<INTEGER> <REAL>]<BASE PART>?

SEMANTICS

A numeric literal may be split across card images. The length of a numeric literal must not exceed the following formula:

$$I + N \leq 62, \text{ if base is decimal}$$

$$I + N \leq 61, \text{ for non-decimal base,}$$

where I represents the total number of integer and fractional digits, not counting the radix point, and N represents the number of digits in the normalized exponent, not counting the exponent delimiter. No blanks may be embedded within a numeric literal. The exponent part may not exceed the range ± 999 . The base part must be in the range 2-36.

EXAMPLES

INTEGER

1

0A3456 (must start with a decimal digit)

FIXED POINT

1.

1.ABCDE

.34291

REAL

@23

1.@43

2489732@-728

77A34Q.9L7@+3

<*N>

ABCDE(16)

3.489@23(12)

1011110001110(2)

<*S>, the String Literal Metaclass

TBNF SYNTAX

<STRING BODY>

::= [<ANY> but [" not "]]*

<STRING QUOTE>

::= "

<*S>

::= <STRING QUOTE><STRING BODY><STRING QUOTE>

SEMANTICS

The definition of <STRING BODY> indicates that if the string quote appears within the body, it must be double. When the string is reduced to a BIGTAB entry, the redundant quote is deleted. The string literal must be completely contained in one card-image, and may not exceed 48 characters (not counting redundant double string quotes).

EXAMPLES

""""INPUT"""" (stored as "INPUT")

"EXPAND", "ILLIAC IV TRANSLATOR WRITING SYSTEM"

THE TWS MACRO FACILITYTBNF SYNTAX

<MACRO DECLARATION>	::= DEFINE <u>list</u> <MACRO DEFINITION> <u>separator</u> ", " ";"
<MACRO DEFINITION>	::= <*I><MACRO FORMAL PARAMETER PART> "=" [<ANY> <u>but</u> <MEND> <u>but</u> DEFINE]* <MEND>
<MACRO FORMAL PARAMETER PART>	::= "[" <u>list</u> <*I> <u>separator</u> ", " "]" "(" <u>list</u> *I <u>separator</u> ", " ")"
<MEND>	::= "?" MEND <CONTROL CARD MEND>
<MACRO INVOCATION>	<*I><MACRO ACTUAL PARAMETER PART>
<MACRO ACTUAL PARAMETER PART>	::= "[" <u>list</u> <MACRO ACTUAL PARAMETER> <u>separator</u> ", " "]" ")" <u>list</u> <MACRO ACTUAL PARAMETER> <u>separator</u> ", " ")"

`<MACRO ACTUAL PARAMETER> ::= [<ANY> but DEFINE but
<UNBRACKETED COMMA>]*
<UNBRACKETED COMMA> ::= ", "`

SEMANTICS

The limitation that a macro definition not contain "DEFINE" enforces the rule that macro declarations not be nested. Macro calls may exist either within the defined text, or in an actual parameter. The restriction exists that a macro may not contain a call on itself, either directly or indirectly.

The unbracketed comma is a recognition that an actual parameter may contain virtually any text. Specifically, if another macro call is in the actual parameter part, or a call on a procedure containing its own actual parameters, delimited by commas, a way must be devised for defining the parameter delimiter comma. The definition that has been implemented is:

`<UNBRACKETED COMMA> ::=` A level zero comma where, when the
initial "[" or "(" is recognized, the
level is set to zero, and incremented
by one for each subsequent "[" or "("
and decremented by one for each sub-
sequent "]" or ")".

This concept could be expanded by modifying the procedure PROCESS-MACROACTUALPARAMETERPART in a language-specific manner to accommodate such additional bracketing pairs that might occur in an actual parameter as
INTEGER - ";" or REAL - ";".

`<CONTROL CARD MEND>` is a control card option "MACROEND X", where X may be "MEND" or a single special character. If "MEND" appears, then this keyword

will mark the end of the macro definition. If a single special character, then that will end the definition.

EXAMPLES

<MACRO DECLARATION>

\$ MACROEND #

DEFINE MACROTABLE(P1) = MACROTAB[(TWSTI P1).[36:5],

TWSTI.[41:7]]#, INCR(X) = X ← X+1 #, ALPHABETIC = [42:6] # ;

MACRO INVOCATION

. . . MACROTABLE(IF A+B Y THEN ELSE (INCR(W))) . . .

ADD (A[3,4,7,9,12], B[4,3]);

APPENDIX B

SAMPLE PROGRAM

In order to illustrate the concepts described in this paper, we consider a comprehensive example. This begins with the TBNF grammar that defined the initial symbol table and proceeds through the balancing of the symbol table at the beginning of the compilation of a program and through the scanning of the text of a sample program.

TBNF Grammar for the Language DEMALGOL

The following language is used as a TWS bench mark. This language, with the addition of the semantic actions, is documented in Trout [4].

DEMALGOL

<PROGRAM>	::= <BLOCK>;
<BLOCK>	::= BEGIN <DECLARATION>* <u>list</u> <STATEMENT> <u>separator</u> ";" END ;
<DECLARATION>	::= [INTEGER BOOLEAN LABEL] <u>[list</u> <*I> <u>separator</u> ", " <ERROR>;
<STATEMENT>	::= [<LABEL> ":"]*[<GO TO GOTO><LABEL> IF <BOOLEAN> THEN <STATEMENT> IF <BOOLEAN> THEN <STATEMENT> [ELSE <STATEMENT>]? <VARIABLE> [":=" "←"] <VALUE> <> [<u>ahead</u> ";" <u>ahead</u> END <u>ahead</u> ELSE] <ERROR>;

BIGTAB INDEX	BIGTAB CONTENTS
15	0/0/0/16
16	0/12/19/21
17	00DEMAG
18	000L
19	66/05/43/26
20	00BEGIN
21	67/03/41/23
22	00END
23	68/11/31/29
24	00INTEGE
25	00R
26	69/11/0/47
27	00B00LEA
28	00N
29	70/05/0/33
30	00LABEL
31	71/02/0/35
32	00GO
33	72/02/39/0
34	00TO
35	73/04/0/37
36	00GOTO
37	74/02/0/0
38	00IF
39	75/04/45/0
40	00THEN
41	76/04/0/0
42	00ELSE
43	77/03/0/0
44	00AND
45	78/02/49/0
46	00OR
47	0/06/0/0
48	00DEFINE
49	0/04/0/0
50	00MEND

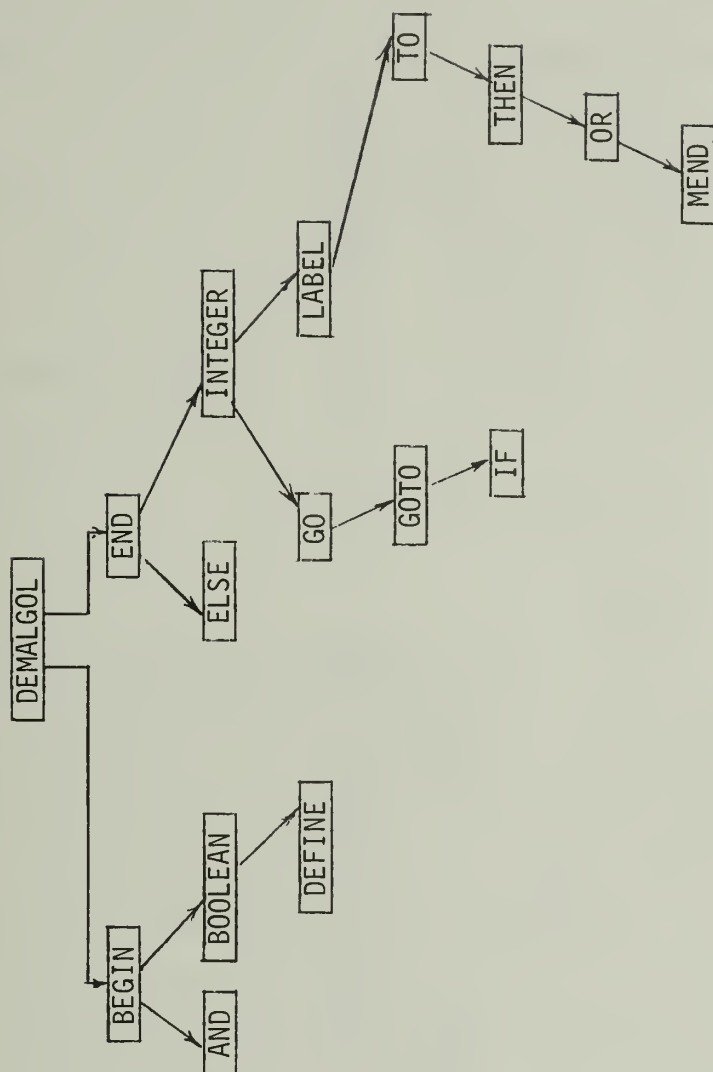


Figure 5. Unbalanced Tree Structure

BIGTAB INDEX	BIGTAB CONTENTS
15	0/0/0/33
16	0/12/0/0
17	00DEMALG
18	000L
19	66/05/43/26
20	00BEGIN
21	67/03/0/0
22	00END
23	68/11/0/0
24	00INTEGE
25	00R
26	69/11/0/0
27	00B00LEA
28	00N
29	70/05/37/45
30	00LABEL
31	71/02/47/29
32	00G0
33	72/02/31/0
34	00T0
35	73/04/0/0
36	00G0T0
37	72/02/73/27
38	00IF
39	75/04/0/0
40	00THEN
41	76/04/16/21
42	00ELSE
43	77/03/0/0
44	00AND
45	78/02/49/39
46	000R
47	0/06/19/41
48	00DEFINE
49	0/04/0/0
50	00MEND

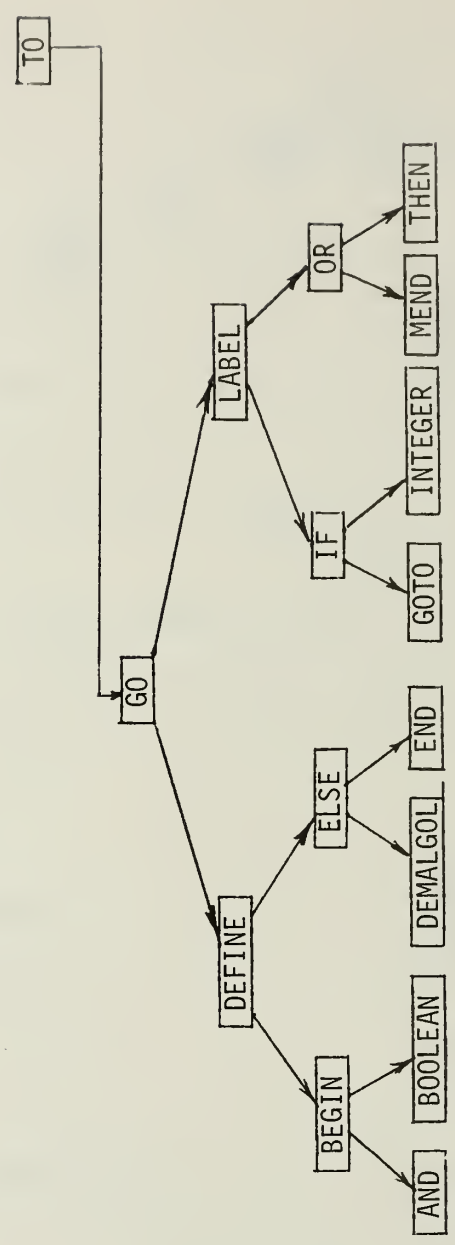


Figure 6. Balanced Tree Structure

ACTION BY THE SCANNER ON THIS PROGRAM

This section will trace the major actions of the scanner on text lines 1-18 of the above program, with the following notes on format:

- a) SCAN descriptors are shown with five fields, representing the class, symbol number or semantic part, BIGTAB pointer or symbol number, class and BIGTAB pointer or symbol number. The values are in decimal.
- b) BIGTAB header words are shown with four fields: semantic part, number of words/characters, left tree pointer, right tree pointer. In the case of the semantic part of macro definitions or parameters, the semantic part is represented by (a/b/c), where a is bit [1:1], the macro flag, b is bit [2:1], the formal parameter flag, and c is bit [3:12], the macro address or parameter number.
- c) BIGTAB data words are shown as they appear in section 2.2, i.e., in character notation.
- d) MACROTAB entries are shown with various fields, as described in section 2.4.3. If a field is designated as unused, it will not be listed below.

\$RSWD	LIST	ALPHANUMERIC - BALANCE	001
--------	------	------------------------	-----

CHARCLASS[#] = 0

SPSTYP = 0 (i.e., reserved word option rather than special)

CHARCLASS[-]. 44 = 1

(See Figure 7 for the results of balancing the tree)

BIGTAB[1] = 33 (links initial balanced BIGTAB to <*I> tree)

BIGTAB[15] = 0

Lines 2, 3 and 4 cause no output by the scanner other than the listing.

BEGIN			005
-------	--	--	-----

*SCAN descriptor	15/66/24/15/66	BEGIN	
------------------	----------------	-------	--

INTEGER A, B, C, X, TEMP			006
--------------------------	--	--	-----

*SCAN descriptor	15/68/34/15/68	INTEGER	
------------------	----------------	---------	--

BIGTAB[51]	=	0/01/0/0	
------------	---	----------	--

BIGTAB[52]	=	00A	
------------	---	-----	--

BIGTAB[43]	=	77/03/51/0	(link to AND)
------------	---	------------	---------------

*SCAN descriptor	1/0/51/1/51	A	
------------------	-------------	---	--

*SCAN descriptor	15/58/58/15/58	,	
------------------	----------------	---	--

BIGTAB[53]	=	0/01/0/0	
------------	---	----------	--

BIGTAB[54]	=	00B	
------------	---	-----	--

BIGTAB[43]	=	77/03/51/53	(link to AND)
------------	---	-------------	---------------

*SCAN descriptor	1/0/53/1/53	B	
------------------	-------------	---	--

*SCAN descriptor	15/58/58/15/58	,	
------------------	----------------	---	--

BIGTAB[55]	=	0/01/0/0	
------------	---	----------	--

BIGTAB[56]	=	00C	
------------	---	-----	--

BIGTAB[26]	=	69/11/0/55	(link to BOOLEAN)
------------	---	------------	-------------------

*SCAN descriptor	1/0/55/1/55	C	
------------------	-------------	---	--

*SCAN descriptor	15/58/58/15/58	,	
------------------	----------------	---	--

BIGTAB[57]	=	0/01/0/0	
------------	---	----------	--

BIGTAB[58]	=	00X	
------------	---	-----	--

BIGTAB[33]	=	72/02/31/57	(link to T0)
------------	---	-------------	--------------

*SCAN descriptor	1/0/57/1/57	X	
------------------	-------------	---	--

*SCAN descriptor	15/58/58/15/58	,	
------------------	----------------	---	--

BIGTAB[59]	=	0/04/0/0	
------------	---	----------	--

BIGTAB[60] = 00TEMP
 BIGTAB[39] = 75/04/59/0 (link to THEN)
 *SCAN descriptor 1/0/59/1/59 TEMP

LABEL GOOD-RIGHT-TRIANGLE, NOT-A-RIGHT-TRIANGLE, 017

*SCAN descriptor 15/70/29/15/70 LABEL
 BIGTAB[61] = 0/31/0/0
 BIGTAB[62] = 00GOOD-R
 BIGTAB[63] = 00IGHT-T
 BIGTAB[64] = 00RIANGL
 BIGTAB[65] = 00E
 BIGTAB[35] = 73/04/61/0 (link to GOTO)
 *SCAN descriptor 1/0/61/1/61 GOOD-RIGHT-TRIANGLE
 *SCAN descriptor 15/58/58/15/58 ,
 BIGTAB[66] = 0/32/0/0
 BIGTAB[67] = 00NOT-A-
 BIGTAB[68] = 00RIGHT-
 BIGTAB[69] = 00TRIANG
 BIGTAB[70] = 00LE
 BIGTAB[59] = 0/04/66/0 (link to TEMP)
 *SCAN descriptor 1/0/66/1/66 NOT-A-RIGHT-TRIANGLE
 *SCAN descriptor 15/58/58/15/58 ,

READ-LOOP 008

BIGTAB[71] = 0/13/0/0
 BIGTAB[72] = 00READ-L
 BIGTAB[73] = 0000P
 BIGTAB[66] = 0/32/0/71 (link to NOT-A-RIGHT...)

*SCAN descriptor	1/0/71/1/71	READ-LOOP	
DEFINE ABS (ARG, ANS) =			009
BIGTAB[74]	=	(1/0/0)/03/0/0	
BIGTAB[75]	=	00ABS	
BIGTAB[51]	=	0/01/0/74	(link to A)
MACROTAB[1]	=	0/74/0/0	(set up head word 2)
BIGTAB[76]	=	(1/1/0)/03/0/0	
BIGTAB[77]	=	00ARG	
BIGTAB[53]	=	0/01/76/0	(link to B)
MACROTAB[4095]	=	0/77	(formal parameter save)
BIGTAB[78]	=	(1/1/1)/03/0/0	
BIGTAB[79]	=	00ANS	
BIGTAB[76]	=	(1/1/0)/03/78/0	(link to ARG)
MACROTAB[4094]	=	0/79	(formal parameter save)
TEMP ← ARG;			010
MACROTAB[2]	=	1/0/59/1/59	(scan of TEMP)
MACROTAB[3]	=	15/31/31/15/31	(scan of ←)
MACROTAB[4]	=	10/0/0	(formal parameter)
MACROTAB[5]	=	15/46/46/15/46	(scan of ;)
IF TEMP < 0 THEN			011
MACROTAB[6]	=	15/74/37/15/74	(scan of IF)
MACROTAB[7]	=	1/0/59/1/59	(scan of TEMP)
MACROTAB[8]	=	15/30/30/15/30	(scan of <)
BIGTAB[77]	=	0/06/0/0	
BIGTAB[78]	=	01000000	
BIGTAB[2]	=	0/0/0/77	(start numeric tree)

MACROTAB[9] = 2/0/77/2/77 (scan of 0)
 MACROTAB[10] = 15/75/39/15/75 (scan of THEN)

ANS ← 0 - TEMP 012

MACROTAB[11] = 10/0/1 (formal parameter ANS)

MACROTAB[12] = 15/31/31/15/31 (scan of ←)

MACROTAB[13] = 2/0/77/2/77 (scan of 0)

MACROTAB[14] = 15/44/44/15/44 (scan of -)

MACROTAB[15] = 1/0/59/1/59 (scan of TEMP)

ELSE ANS ← TEMP ? , 013

MACROTAB[16] = 15/76/41/15/76 (scan of ELSE)

MACROTAB[17] = 10/0/1 (formal parameter ANS)

MACROTAB[18] = 15/31/31/15/31 (scan of ←)

MACROTAB[19] = 1/0/59/1/59 (scan of TEMP)

MACROTAB[20] = 15/14/14/15/14 (scan of ?)

BIGTAB[78] = 0/03/0/0 (reset BIGTAB semantic-ARG)

BIGTAB[76] = 0.03/0/0 (reset BIGTAB semantic-ANS)

MACROTAB[0] = 20/0/2 (set up head word 1)

GO-TO-GOOD = 014

BIGTAB[79] = (1/1/21)/14/0/0

BIGTAB[80] = 00GO-TO-

BIGTAB[81] = 00GOOD

BIGTAB[61] = 0/31/79/0 (link to GOOD-RIGHT...)

MACROTAB[22] = 0/79/0/0 (set up head word 2)

IF X < 0.1 THEN GO TO GOOD-RIGHT-TRIANGLE ? ; 015

MACROTAB[23] = 15/74/37/15/74 (scan of IF)

MACROTAB[24] = 1/0/57/1/57 (scan of X)

MACROTAB[42]	=	15/32/32/15/32	(scan of x)
MACROTAB[43]	=	1/0/53/1/53	(scan of B)
MACROTAB[44]	=	15/45/45/15/45	(scan of))
MACROTAB[45]	=	15/45/45/15/45	(scan of))
MACROTAB[46]	=	14/44/44/15/44	(scan of -)
MACROTAB[47]	=	15/29/29/15/29	(scan of ()
MACROTAB[48]	=	1/0/55/1/55	(scan of C)
MACROTAB[49]	=	15/32/32/15/32	(scan of x)
MACROTAB[50]	=	1/0/55/1/55	(scan of C)
MACROTAB[51]	=	15/45/45/15/45	(scan of))
MACROTAB[52]	=	15/58/58/15/58	(scan of parameter delimiter comma)
MACROTAB[32]	=	19/33/0/0	(actual parameter addresses and lengths)
MACROTAB[53]	=	1/0/57/1/57	(scan of X, parameter 2)
MACROTAB[54]	=	15/45/45/15/45	(scan of), end of parameter part)
MACROTAB[32]	=	19/33/1/53	(actual parameter addresses and lengths)
MACROTAB[20]	=	9/0/0	(set outermost return descriptor)
*SCAN descriptor		1/0/59/1/59	(TEMP, from MACROTAB[2])
*SCAN descriptor		15/31/31/15/31	(, from MACROTAB[3])
MACROTAB[52]	=	9/5/0	(set parameter 1 return)
*SCAN descriptor		15/29/29/15/29	((, from MACROTAB[33], actual parameter table)
	.		
	.		
	.		

*SCAN descriptor	15/45/45/15/45	(), from MACROTAB[51])
*SCAN descriptor	15/46/46/15/46	(;, from MACROTAB[5])
	.	
	.	
	.	
*SCAN descriptor	15/75/39/15/75	(THEN, from MACROTAB[10])
MACROTAB[54] =	9/12/0	(set parameter 2 return)
*SCAN descriptor	1/0/57/1/57	(X, from MACROTAB[53], actual parameter table)
*SCAN descriptor	15/31/31/15/31	(←, from MACROTAB[12])
	.	
	.	
	.	
*SCAN descriptor	15/76/41/15/76	(ELSE, from MACROTAB[16])
MACROTAB[54] =	9/18/0	(parameter 2 return word)
*SCAN descriptor	1/0/57/1/57	(X, from MACROTAB[53])
*SCAN descriptor	15/31/31/15/31	(←, from MACROTAB[18])
*SCAN descriptor	1/0/59/1/59	(TEMP, from MACROTAB[19])
*SCAN descriptor	15/46/46/15/46	(;, from card 17)
GO-TO-GOOD;		018

MACROTAB[31] =	9/0/0	(set outermost return)
*SCAN descriptor	15/74/37/15/74	(IF, from MACROTAB[23])
	.	
	.	
	.	
*SCAN descriptor	1/0/61/1/61	(GOOD-RIGHT-TRIANGLE, from MACROTAB[30])
*SCAN descriptor	15/46/46/15/46	(;, from card 18)

LIST OF REFERENCES

- [1] Abel, N. A., et al, "TRANQUIL: A Language for an Array Processing Computer," Proceedings of the Spring Joint Computer Conference, 1969.
- [2] Lawrie, D., "GLYPNIR. A List Processing Language for Illiac IV," Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 322, April 1969.
- [3] Feldman, J., and D. Gries, "Translator Writing Systems," Communications of the ACM, Vol. 11, No. 2, pp. 101-113, February 1968.
- [4] Trout, H. R. G., "A BNF-like Language for the Description of Syntax Directed Compilers," Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 300, January 1969.
- [5] Machado, N. C., "ISL--A Semantics Language for a Translator System," M.S. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 367, December 1969.
- [6] Knuth, D. E., The Art of Computer Programming. Volume 1, Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.
- [7] Mercer, D., "TWINKLE--A Syntax Language for a Translator Writing System," M.S. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 396, May 1970.
- [8] Burroughs B-5500 Information Processing Systems Reference Manual, Burroughs Corporation, Detroit, Mich., September 1968.
- [9] Organick, E. I., A MAD Primer, published by the author, 1964.



BIBLIOGRAPHIC DATA SHEET		1. Report No. UIUCDCS-R-73-596	2.	3. Recipient's Accession No.	
4. Title and Subtitle A GENERALIZED LEXICAL SCANNER FOR A TRANSLATOR WRITING SYSTEM			5. Report Date October 1973		
			6.		
7. Author(s) Albert Cannon Baker, Jr.			8. Performing Organization Rept. No. UIUCDCS-R-73-596		
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801			10. Project/Task/Work Unit No.		
			11. Contract/Grant No. US NSF-GJ-596		
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C. 20550			13. Type of Report & Period Covered Master's Thesis		
			14.		
15. Supplementary Notes					
6. Abstracts This is an expository paper that is concerned with a Lexical Scanner for a translator writing system that has been in use at the University of Illinois. Its significant features include a structured, binary-tree symbol table, a parameterized macro expander, and a compile-time flexibility for assigning characters that make up the basic terminal symbols. A comprehensive example of the scanner's operation is also included.					
7. Key Words and Document Analysis. 17a. Descriptors lexical scanner, translator writing system, compiler-compiler.					
b. Identifiers/Open-Ended Terms					
1c. COSATI Field/Group					
1. Availability Statement UNLIMITED		19. Security Class (This Report) UNCLASSIFIED		21. No. of Pages 61	
		20. Security Class (This Page) UNCLASSIFIED		22. Price	

JAN 14 1974

APR 4 1974



UNIVERSITY OF ILLINOIS-URBANA
510.64 IL6R no. C002 no. 595-600(1973)
Design of high noise immunity analog tre



3 0112 088400954